SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **00**:1–33 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

JAC: An aspect-based distributed dynamic framework

Renaud Pawlak [1], Lionel Seinturier [1,2],
Laurence Duchien [1,*,†], Gérard Florin [3],
Fabrice Legond-Aubry [2,3], Laurent Martelli [4]

[1] *Lab. LIFL & INRIA, GOAL/Jacquard, Bâtiment M3, 59655 Villeneuve d'Ascq, France*
[2] *Lab. LIP6, SRC, 4 place Jussieu, 75252 Paris, France*
[3] *Lab. CEDRIC-CNAM, 292 rue Saint Martin, 75141 Paris, France*
[4] *AOPSYS, 5 rue Brown Séquard, 75015 Paris, France*

SUMMARY

In this paper, we present the JAC (Java Aspect Components) framework for building aspect-oriented distributed applications in Java. This paper describes the aspect-oriented programming model and the architectural details of the framework implementation. The framework enables extension of application semantics for handling well-separated concerns. This is achieved with a software entity called an aspect component (AC). ACs provide distributed pointcuts, dynamic wrappers and metamodel annotations. Distributed pointcuts are a key feature of our framework. They enable the definition of crosscutting structures that do not need to be located on a single host. ACs are dynamic. They can be added, removed, and controlled at runtime. This enables our framework for use in highly dynamic environments where adaptable software is needed.

KEY WORDS: Aspect-oriented programming, distribution, dynamicity

1. Introduction

In distributed environments, applications run in an open context. They use networks and their associated services where quality of service is not always guaranteed and may change quickly. In these environments, several concerns must be considered including: fault-tolerance, data consistency, remote version update, runtime maintenance, dynamic lookup, scalability, lack of rate. Addressing these issues may require dynamic and fast reconfiguration of distributed

*Correspondence to: Pr. L. Duchien, USTL - Lab. LIFL, Batiment M3, 59655 Villeneuve d'Ascq, France
†E-mail: duchien@lifl.fr

Received
Revised

applications. Separation of concerns (SoC) and Aspect-Oriented (AO) approaches can provide some answers.

Separation of concerns (SoC) has its roots in the well-known software engineering *divide-and-conquer* principle: it is easier to distinguish, capture and model the function of small software entities rather than working on the entire application [1, 2]. A concern is any functionality that needs to be incorporated in an application. Concerns are isolated during the analysis phase and, depending on the application, a concern may be functional or non-functional. In distributed systems, many concerns need to be incorporated. The interactions between these concerns tend to be numerous and complex. Moreover, these interactions can not be cleanly localized in a single class. For instance, calls to a logging API are spread throughout the whole application. Thus, concerns that can not be cleanly localized in a single class or modular unit, are said to be crosscutting concerns.

Crosscutting concerns lead to implementations where the code is said to be tangled. Classes mix several concerns, and modularity and SoC are lost. To regain these properties, different approaches have been proposed: Aspect-Oriented Programming [3], Composition Filters [4, 5], Aspectual Components [6], Subject-Oriented Programming [7, 8, 9], or Multi-Dimensional Separation of Concerns [10, 11]. In this article, we focus on Aspect-Oriented Programming (AOP).

AOP has been pioneered by Gregor Kiczales and his team at Xerox PARC. They define the notion of aspect as being a software entity that implements a crosscutting concern. For instance, security, which is in many applications a crosscutting concern, can be implemented as an aspect. The AspectJ [12, 13] language is the most widely known aspect compiler for Java. Many other AOP languages or frameworks exist for Java, and for other programming languages. Section 6 reviews some of them. The notion of aspect is quite general, and as classes, aspects may be involved in all the steps of the software life cycle. Until now, many research works put the emphasis on aspects for programming languages. Nevertheless, aspects oriented design is also a topic of interest. For instance, [14] and [15] propose approaches to identify and design aspects. [16], [17], [18] define UML-based design notations for aspects.

This article provides insight on the JAC framework [†] (Java Aspect Components) [19, 20]. JAC goal is to provide a set of concepts to enable distributed and dynamic AOP. Among other topics, we provide support for runtime manipulation of aspects within a distributed environment. Section 2 recalls some AOP definitions and concepts. Section 3 presents the JAC programming model and depicts the semantics of all the available programming concepts. Section 4 proposes practical programming samples of aspects, including distributed aspects and dynamic reconfiguration. Section 5 gives an overview of JAC architecture. Section 6 compares our approach with other projects. Finally we conclude and present future directions.

[†]Available for download under LGPL from `http://jac.objectweb.org`

2. Aspect-oriented programming concepts and definitions

This section briefly recalls the concepts and definitions of AOP. Readers interested in a more thorough description should refer to [3], [21] or [22, 23].

Object-oriented languages provide classes as units of modularity for concerns. However, some concerns such as persistence or security, can not be neatly encapsulated into classes because they affect multiple classes. Such a concern is said to be crosscutting, because it must be implemented as part of multiple classes in a manner that crosscuts program structure. Hence, developing, debugging, or modifying a crosscutting concern with an object-oriented approach require to modify all the classes involved in the crosscutting. In many cases, this is too error prone and counter-productive. Aspects have been introduced to modularize crosscutting concerns. They introduce the following notions: join point, pointcut, and advice. Aspects are sets of pointcuts and advices.

A joint point is a point in the execution of a program: for instance, a method call, a method execution, a field read, a field write, or catching an exception. Each AO language or framework defines its list of supported join points. The method execution join point is the most standard one and is supported by most of the known approaches.

A pointcut expresses a crosscutting policy and is defined as a set of join points: they are the points where the aspect modifies the program. All known AO languages and frameworks use regular expressions, i.e. expressions with logical operators and wildcards, to define pointcuts. This allows to capture in a single expression several join points.

An advice is a piece of code that is associated to a pointcut. It defines instructions to be run each time one of the join points designated by the pointcut is encountered. In most of the known approaches, advices can be run before, after, or around (i.e. both before and after, with the ability to explicitly tell when the join point has to be executed) the join point.

AOP also introduces the notion of a weaver. Weaving is the process of taking as input a program and some aspects, and producing as output a program or a program execution, modified by the aspects. Weaving can be static (e.g. performed at compile time), or dynamic (e.g. performed at run time). The weaver is the program that performs this weaving process.

3. The JAC programming model

This section presents the programming concepts provided by JAC for building dynamic and distributed aspects. JAC is a programming framework, not a new language. Manipulation of concepts is achieved by extending the classes of the framework and by using the JAC API. There are two levels of AOP with JAC:

- The programming level where one can program totally new aspects. At this level, which is the same as AspectJ, programmers create new aspects, new pointcuts, and new wrappers to implement crosscutting concerns. This level is presented in sections 3.1 and 3.2.
- The configuration level where one can customize existing aspects to make them work with existing applications. This level is supported by a configuration language with a generic syntax that allows the programmer to call configuration methods on existing aspects. In

the JAC philosophy, it is very important to understand that everything is done to avoid the programming of new aspects when not needed. Thus, we focus on aspect reusability and we furnish a set of aspects with ready-to-use configuration methods for that. This level is presented in section 3.3.

3.1. Aspect components

Aspect Components (ACs for short) are the central notion of our AO framework. Much like the components that can be found in other programming models (see for instance the Aspectual Components model [6]), our ACs are the implementation units that define extra characteristics which crosscut a set of business objects (also called base objects). These components are hosted by JAC containers [24]. A container provides a runtime context for applications consisting of ACs' instances and business objects. Much like containers for EJB [25] components, JAC containers are remotely accessible servers. But contrary to EJB containers that only host business components, JAC containers host both business components and ACs' instances. Hence, technical concerns such as persistence, transactional support, or security are not restricted to the ones hard coded in the container, but new concerns can be plugged into the container. An application is a collection of base and aspect objects that may be distributed on several remote containers. We thus provide a straightforward means to develop distributed applications based on distributed aspects.

3.1.1. Programming aspect components

In JAC, a new aspect is defined by subclassing the `jac.core.AspectComponent` class. This class provides default primitives to implement technical requirements that influence a set of functional objects in a systematic way. More specifically, this class provides an API to define pointcuts. Pointcuts are associated to wrapping classes and methods (see section 3.2) that can add extra treatments before/after/around base methods on a per-object basis. Contrary to AspectJ that deals with aspect on a per-class basis, instances from the same class in JAC may be modified differently. This feature is particularly useful in distributed environments where different policies for persistence or security need to be applied to different server objects, that are instances of the same class. The semantics of pointcut expressions is explained below.

```
import jac.core.AspectComponent;

public class MyAspect extends AspectComponent {
  public MyAspect() {
     pointcut(    // define a pointcut
        "*",                 // object expression
        "org.jac.MyClass",   // class expression
        "get():int",         // method expression
        "renaud.jac.com",    // container name expression
        MyWrapper.class,     // wrapping class
        "wrappingMethod"     // wrapping method
```

```
        );
    }
}
```

3.1.2. Pointcuts semantics

A pointcut is attached to an AC and conversely, an AC can manage several pointcuts. A pointcut is composed of four pointcut expressions matching respectively, some objects, classes, methods, and hosts. The idea is to filter out the sets of all possible methods, given an application composed of objects deployed on some hosts. This filtering defines the points where ACs modify or enhance the behavior of the application. Pointcuts are expression with wildcards (`"?"` for any single character or `"*"` for a set of characters) extended by keywords for method filtering (see table I). A given object/class/method/host is extended by the pointcut if:

- the first expression (the object pointcut expression) matches the name of the object (any object hosted by a JAC container is named as explained in section 3.1.3),
- the second expression (the class pointcut expression) matches the class name,
- the third expression (the method pointcut expression) matches the method prototype (for instance, to match a method `"void get(int i)"`, the expression is `"get(int):void"` or any expression that matches this string – note that regular expressions can also be used in this context),
- the last expression (the host pointcut expression) matches the container name where the owner object is located. Whenever a wildcard expression is specified, the set of all hosts (i.e. `"*"`) is defined as the set of all hosts specified in the deployment descriptor of the application.

Keywords are also provided in pointcut expressions to designate method sets (e.g. the set of setter methods for a given field). Beyond simplifying pointcut writing, keywords let the pointcut definition be independent from method names and from specific naming conventions (e.g. a setter method name does not have to begin with set). Each set of methods associated with a keyword is computed by the JAC runtime during a bytecode analysis phase (see section 3.4). Table I shows some of the available keywords (see the appendix for all the full list). These keywords greatly simplify the expressions so that the use of wildcards or regular expressions is not required most of the time. Furthermore, pointcut sub-expressions can be composed using logical AND (&&), OR (||) and NOT (!) operators.

3.1.3. Naming semantics

Object naming can be useful in several cases. For instance, a persistence framework may need to define objects as persistent roots from where other referenced objects are persistent. A multi-user application may want to exchange data between several users by using specific objects. In a distributed environment, object names are almost mandatory. They are the basis on which a client object binds itself to a server object before invoking one of its methods. For instance,

Table I. Keywords in pointcut expressions (see the appendix for the full list)

Keyword	Semantics
ALL	all methods (alias for *)
GETTERS[(*name*)]	getter methods for *name* (all getters if no name is specified)
SETTERS[(*name*)]	setter methods for *name* (all setters if no name is specified)
MODIFIERS[(*name*)]	methods that modify *name* (all modifiers of the object fields if no name is specified)
ACCESSORS[(*name*)]	methods that access *name* (all accessors for the object fields if no name is specified)
CONSTRUCTORS	all contructors

RMI server objects are named with a URL (strings such as `rmi://some.host/aLocalName`), CORBA objects are named with IORs (Interoperable Object References) or URLs (strings such as `corbaname::some.host#aLocalName`), Web Services are named with HTTP URLs. As the goal of JAC is to develop AO applications for these distributed systems, a naming policy has been included in the JAC framework as an aspect.

The naming aspect operates on a pointcut that matches the object's constructors and registers the objects within a repository. These objects can be retrieved by the program or the aspects at any time. Moreover, these names can be used within the pointcut definitions (in the object expression) as described in section 3.1.2.

By default, objects are associated with a local name built from their class name followed by their instance number that corresponds to the instance creation order (e.g. `aclass0` for the first instance of class `aClass`). This naming policy can be tuned and overridden to provide customized names. Whenever the object is involved in a distributed application, this local name is used to construct a distributed name (e.g. `rmi://some.host/aclass0` where `aclass0` is a RMI object located on `some.host`).

3.2. Dynamic wrappers

Dynamic wrappers [24] in JAC are the software entities that allow the modification or enhancement of the semantics of some base objects. Like advice in AspectJ, they provide the ability to add *before*, *after*, or *around* code on existing methods. A dynamic wrapper is defined as a regular stand-alone object (i.e. it contains fields that form its state, and methods that define its functionalities). We say that the base object is wrapped by a wrapper. The base object is then called a wrappee. Wrappers are dynamic in JAC: they can be added or removed on the fly, while the application is running.

3.2.1. Programming dynamic wrappers

Dynamic wrappers are regular Java classes extending the `java.core.Wrapper` class. In addition to regular methods, dynamic wrappers can contain three kinds of methods.

- *wrapping methods*: perform treatments before, after and around the methods of a base object (same as the *around* advice in AspectJ). A wrapping method wraps one or several methods (defined in one or several base objects).
- *role methods*: extend regular objects interfaces (similarly to the *introduce* statement in AspectJ).
- *exception handlers*: handle exceptions raised by called objects in the object the wrapper is applied to.

The following code shows a wrapper containing the three kinds of methods.

```
    import jac.core.Wrapper;
    import jac.core.AspectComponent;
    import jac.core.Interaction;
    import jac.core.Wrappee;

    public class MyWrapper extends Wrapper {
      public MyWrapper( AspectComponent ac ) {super(ac);}

1>    public Object aWrappingMethod( Interaction interaction ) {
        /* ... some before code ... */
2>      Object ret = proceed(interaction);
        /* ... some after code ... */
        return ret;
      }
3>    public Object doNothingWrappingMethod( Interaction interaction ) {
        /* Introspect the current interaction */
        Wrappee wrappee = interaction.wrappee;
        jac.core.rtti.AbstractMethodItem method = iteraction.method;
        Object[] arguments = iteraction.args;
        return proceed(interaction);
      }

4>    public Object aRoleMethod(
        Wrappee wrappee /* , ... any other parameters ... */ ) {
        /* ... some code ... */
      }

5>    public void anExceptionHandler(
        Interaction interaction, AnException e ) {
        /* ... some code ... */
```

```
} }
```

In JAC, the joinpoint between a wrappee and a wrapper can be introspected with the `jac.core.Interaction` API. An interaction instance provides methods and public fields to retrieve: (i) the wrappee reference (field `wrappee`), (ii) the wrapped method (field `method`), (iii) the arguments of the call (field `args`). The type of the `method` field is explained later in section 3.4. Line 3 of the above code shows a wrapping method that simply accesses these fields. Their values are set by the JAC runtime whenever a call for the wrapped method is issued.

Methods with only one parameter of type `Interaction` and returning an `Object` are qualified as wrapping methods. Lines 1 and 3 of the above code show examples of wrapping methods. After performing some before code (or not, if there is no relevant before code), the wrapper delivers the call by calling `proceed(Interaction)`, inherited from `Wrapper`. In simple cases, this delivery consists of calling the wrapped method. Nevertheless, several aspects can be woven into a same base object leading to several wrappers wrapping the same method. We then say that there is a wrapping chain. This situation raises a composition issue that we discussed in earlier papers [26, 20]. This issue is beyond the scope of the current paper: basically we define composition rules that specify the order in which the wrappers must be applied to the base object. The `proceed` method delivers the call to the next wrapper in the chain, or to the wrapped method if there is no more wrappers in the chain. When this call returns, the code after the call to `proceed` is executed in all the wrappers. Note that the call to `proceed` is optional: a wrapper may decide to return from the call without delivering it to the base object.

Methods whose first parameter is of type `Wrappee` are role methods. Line 4 gives an example of a role method. Methods with a parameter of type `Interaction` and a second parameter that is a subclass of `java.lang.Exception` are exception handlers. They catch any matching exception in the execution flow of the wrapper in which they are defined. Line 5 illustrates the definition of an exception handler.

3.2.2. Dynamic wrappers and aspect components

Developing an Aspect Component with JAC requires: (i) defining pointcuts, (ii) associating them with wrapping methods defined in a wrapper classes. By default, on a given host, an instance of the wrapper is associated with all the methods designated by the pointcut. If the pointcut encompasses several hosts, an instance is created on each host. This default behavior can be overridden to associate, for instance, a wrapper instance with each wrapped method.

The following code shows the implementation of the `CoordinatorAC` AC that defines a synchronization policy: all the methods designated by the pointcut will be mutually exclusive (at most one thread may execute the code of these methods). The AC defines a pointcut for all the methods of all the instances of classes in package `org.jac` and recursively in its sub-packages. The pointcut is associated with the wrapping method `mutexclusive` defined in class `CoordinatorWrapper` (an instance of `jac.core.Wrapper` could also have been transmitted instead of `java.lang.Class`). This method is synchronized and simply proceeds the call to the base object.

```
import jac.core.Wrapper;
import jac.core.AspectComponent;
import jac.core.Interaction;

public class CoordinatorAC extends AspectComponent {

  public CoordinatorAC() {
     pointcut(    // define a new pointcut expression
        "*",                          // all instances
        "org.jac.*",                  // all classes of package org.jac
        "*(*):*",                     // all methods
        "renaud.jac.com",             // container renaud.jac.com
        CoordinatorWrapper.class,     // wrapping class
        "mutexclusive"                // wrapping method
     );
  }

  public class CoordinatorWrapper extends Wrapper {
     public CoordinatorWrapper( AspectComponent ac ) {super(ac);}
     public synchronized Object mutexclusive( Interaction i ) {
        return proceed();
     }
  }
}
```

The pointcut matches host `renaud.jac.com`. Note that we may want to provide a more generic pointcut such as `*` (i.e. all the hosts where the application is deployed). In such a case, an instance of `CoordinatorWrapper` would be instantiated on each host, hence synchronizing the base objects of these hosts. Achieving a distributed synchronization scheme is not as straightforward and requires additional programming.

3.3. Configuring aspect components

When implementing distributed applications, most of the technical concerns (security, persistence, transactional support, replication, ...) are crosscutting concerns. By modularizing them in well-identified software entities, AOP clearly provides some benefits in terms of development and maintenance. Nevertheless, codifying concerns is not sufficient. Almost all distributed applications require some significant configuration steps to map the properties of technical concerns onto the business code. For instance, developers need to specify which methods should be transactional, which methods should be restricted to authenticated users, or which data should be persistent. In component models such as EJB [25], these configurations are specified in XML files. The associated DTD are defined in the EJB specifications, and are specific to the technical concerns provided by this application server.

Since these configuration steps are almost mandatory for distributed applications, and since JAC provides the ability to define new aspects for distributed applications, we also have to provide a convenient way to define configuration policies for new aspects. Note also that this configuration mechanism is a key feature for ensuring that libraries of aspects, such as the ones provided with JAC, are re-used efficiently.

Hence, each AC in JAC is associated with a configuration file. This is a text file that is parsed when the AC is instantiated. This file can be though of as an initialization script for the AC. Thus, changing the configuration does not require recompiling the AC. This configuration file is a list of calls to the public methods of the AC. The idea is that each AC defines the methods that are relevant for its configuration. We then obtain a list of commands that defines a kind of Domain Specific Language (DSL) for the configuration of this aspect.

A configuration file is a text file with one call to a public method of the AC class per line. Each line begins with the name of the method to call. Arguments of the call follow, separated with spaces (no parenthesis, no comma). String arguments with spaces or other separators need to be put between double quotes. Array arguments are put between curly brackets and each element is separated by a comma.

As a simple example of the configuration mechanism, we can extend the AC of the previous section by adding a `synchro` method. Given two parameters, this method defines a pointcut for a class and a method.

```
public class CoordinatorAC2 extends CoordinatorAC {

   public void synchro( String cl, String meth ) {
     pointcut(    // define a new pointcut expression
        "*",                       // all instances
        cl,                        // all classes specified by cl
        meth,                      // all methods specified by meth
        "renaud.jac.com",          // container renaud.jac.com
        CoordinatorWrapper.class,  // wrapping class
        "mutexclusive"             // wrapping method
     );
   }
}
```

Next, a configuration file specifies which methods should be synchronized. For instance, the following `CoordinatorAC2.acc` (`.acc` stands for Aspect Component Configuration and is the standard extension for configuration files in JAC) file specify that all methods `write` in all the classes beginning with `C` in `org.jac.samples`, and the method `update` in `org.jac.samples.Example` are synchronized.

```
synchro "org.jac.samples.C*" "write(*):*"
synchro "org.jac.samples.Example" "update():void"
```

3.4. Run-time type information

To weave crosscutting concerns, an aspect weaver has to perform some kind of reflection on the business code. For instance, classes have to be introspected at compile time, load time or run time, to decide whether a method has to be wrapped or not, or whether a field has to be persistent or not. Such decisions are taken according to the pointcut definitions that specify sets of methods modified by aspects.

In some cases, expressions on method names are not the best way to designate such sets, because the expressions are too implementation dependant. For such cases, JAC provides a way to tag methods with attributes that will be looked up by aspects later on to determine whether they have to be applied or not. Such a process is similar to annotations found in numerous extensions of C++ (for parallelism or distribution), in Fortran HPF, in the pragmas for compilation directives, or in the early implementations of meta-object protocols (see for instance OpenC++ v1 [27]).

To achieve this, some data concerning the business code is stored in memory and constructed while the application is loaded. This data provides a representation of classes, of their inheritance and implementation relations, of their fields, and of their methods. An API (called RTTI for Run-Time Type Information) is provided to navigate and query this structure. All the elements of this structure can be annotated with pairs of objects: the first one is a key, the second one is a value. For instance, the following AC (but it could be any other piece of code) queries the RTTI API for method `m` of class `AClass` and attaches to it the key `"forsynchronization"` with the value `true`. This "meta"-data will be used by a wrapper to decide if a method needs to be synchronized, depending on whether this key is set.

```
import jac.core.rtti.ClassRepository;
import jac.core.rtti.ClassItem;
import jac.core.rtti.MethodItem;

public class MyAspect extends AspectComponent {
  public MyAspect() {
     ClassItem cl = ClassRepository.get().getClass(AClass.class);
     MethodItem m = cl.getMethod("m():void");
     m.setAttribute("forsynchronization",new Boolean(true));
  }
}
```

The RTTI API can be thought as the `java.lang.reflect` API with annotations. There are no JAC specific features in it (i.e. the RTTI API is independent and can be used in other projects), but the JAC runtime relies on it to tag methods as setters, getters, or modifiers. These annotations are then queried when defining pointcuts (see section 3.1.2). A detailed description of the API is beyond the scope of this paper. Readers interested in learning more should refer to [28].

```
public class Account {
   String owner;
   double balance = 0.0;

   public Account(String owner) { this.owner=owner; }
   public void credit(double toCredit) { balance += toCredit; }
   public void withdraw(double toWithdraw) { balance -= toWithdraw; }
   public double balance() { return balance; }

   public static void main( String[] args ) {
     Account bob = new Account("Bob");
     bob.credit(100.0);
     bob.widthdraw(50.0);
   }
}
```

Figure 1. A simple business class.

4. Programming with JAC

The goal of this section is to provide a deeper insight on the JAC programming model by showing concrete programming examples. In section 4.1, we show a wrapper that implements constraint checking. Section 4.2 presents a full application constructed by re-using existing ACs. Section 4.3 addresses distributed programming with JAC. Finally, section 4.4 deals with dynamic adaptability.

4.1. A simple aspect

Figure 1 shows a business class that manages bank accounts with methods to add or withdraw a given amount from the owner's account. A `main` method is provided to create an account and perform operations on it.

Figure 2 shows how to implement some constraint checking in a very simple way. This example shows how to use the `jac.core.Interaction` class. `TestAC` checks the bounds of a field, prevents it from being negative, and raises an exception each time an operation tries to do so. The AC performs the following steps:

- line 1: the pointcut denotes method `withdraw` of class `Account` and wraps it with method `limit` defined in class `LimiterWrapper`,

```
  import jac.core.Wrapper;
  import jac.core.AspectComponent;
  import jac.core.Interaction;

  public class TestAC extends AspectComponent {
    public TestAC() {
1>    pointcut( "*", "Account", "withdraw", "*",
                LimiterWrapper.class, "limit" );
    }

    public class LimiterWrapper extends Wrapper {
      public Object limit( Interaction i ) {
2>      Account base = (Account) i.wrappee;
3>      double argAmount = ((Double) i.args[0]).doubleValue();
4>      if ( argAmount > base.getBalance() )
          throw new Exception("<0 forbidden");
5>      return proceed(i);
  } } }
```

Figure 2. A bound checker aspect component.

- line 2 & 3: within the wrapping method, the `Interaction` instance provides access to the wrappee, and to the arguments of the call as an array of objects. The value of the `toWithdraw` parameter is the first and only one in the array.
- line 4: the test. If the argument is greater than the current balance, the call is rejected and an exception is thrown.
- line 5: else, we can proceed by delivering the call to the base object.

As stated in section 3.3 each AC is associated with a configuration descriptor (a .acc file). For this example, no particular configuration is needed and the file is simply empty.

Once the business logic and the ACs have been programmed, developers have to write an application descriptor. This is a simple text file with the .jac extension used by the JAC runtime to launch the application. It contains the main method class name, the class names of the ACs, whether these ACs are woven at start time, and some additional global properties (e.g. the deployment topology that includes the list of hosts on which the application is deployed). A graphical console is provided to unweave or to re-weave aspects at runtime. The following file gives an example of such a file for the previous application.

```
// account.jac
applicationName: bank
```

Table III. Some useful re-usable Aspect Components

aspect name	used to...
naming	name the objects at construction time.
tracing	switch on/off some verbose mode on programs, it can also count the methods invocations that occur in the program.
persistence	define which objects of the application will be stored in a database using JDBC
authentication	restrict the access of some methods to some users
user	handle users and their profiles (often co-used with authentication)
session	make some useful contextual data persistent so that it will remain at the same value for a set of interactions that belong to the same user
GUI	define some presentation information
deployment	define how the objects of the application are deployed on the remote JAC servers defined by the topology when JAC runs in distributed mode
consistency	introduce some protocols to make several remote objects (of the same name) consistent
broadcasting	introduce some protocols to perform broadcasting on remote objects
load-balancing	introduce some protocols to perform load-balancing on remote servers

```
launchingClass: Account
aspects: test TestAC true
```

4.2. Programming an entire application

Besides providing a way to program some customizable aspects, JAC comes with a library of ready-to-use aspects. As a consequence, programming an AO application often just requires choosing the right aspects and configuring them so that they behave properly for the targeted program.

Each existing aspect corresponds to an AC (see section 3.1). Each AC is documented so that a programmer is able to use it just by examining its API (i.e. the set of available configuration methods). Table III gives a list of some of the more useful ACs provided by JAC.

Later we will show (for the impatient reader, see figure 9, page 22) the global control flow between all the elements of the JAC framework for a given application. For the programmer that just uses existing aspects and configures them by writing configuration files, all the underlying mechanisms are transparent.

In order to examine this process, we will use the `Account` class example defined in section 4.1 and we add support for 3 aspects: GUI, persistence and authentication. All the commands used are configuration methods defined in the corresponding AC interfaces.

Figure 3 shows the configuration file for the GUI aspect. The JAC GUI is generic and to introspect on the application objects to provide default views on them. However, with simple reflection, the GUI does not provide all the information that is needed when constructing

```
// make JAC generate all the resolvable parameter names
// for the methods that follow the naming conventions
// (i.e. setter, adders, or removers)
generateDefaultParameterNames Account;

// credit does not follow the naming conventions, we force
// the name that the GUI will use
setParameterNames Account credit {"Amount to be credited"};
// set a default value when credit is called
setDefaultValues Account credit {1};

// same for withdraw
setParameterNames Account withdraw {"Amount to be withdrawn"};
setDefaultValues Account withdraw {1};
```

Figure 3. The gui.acc configuration file.

GUI views. For this reason, the GUI aspect furnishes a set of configuration methods that extend application class semantics with useful data for the GUI. Typical examples of added information are the parameter names of the methods which are not available with the *java.lang.reflect* package. Hence, the *generateParameterNames* and the *setParameterNames* allow the programmer to indicate these names. As shown in figure 6, these indications are provided by RTTI attributes (as depicted in section 3.4).

Other types of configuration can be made with the GUI such as the default parameter values when a method is called. Here, the pop-up window that is opened by the GUI when the *add* method is called will propose the "1" default value to the user, as defined by the *setDefaultValues* configuration method. These are very simple configurations. For more details on all the available configuration methods (such as providing specialized editors or viewers for the fields, changing the default placements in the views, and so on), one can refer to the JAC API at [28].

Figure 4 shows the configuration file of the authentication aspect. The authentication provided by JAC is quite simple and would need some enhancements to be fully usable. Still, it allows the programmer to define some users with their passwords and to restrict some methods. The right to call a method is granted only if the pair *(username, password)* is in the trusted users list. Again, the RTTI is used by this aspect to denote the restricted methods, as shown in figure 6. This aspect also installs some wrappers on the restricted methods and interacts with the GUI to open a pop-up dialog that requires the user to fill the authentication information.

The persistence aspect provided by JAC enables the programmer to make any object or class persistent. A sample use is given in figure 5. A root object is an object that is a persistence

```
// define the users table with their passwords
addTrustedUser "renaud" "renaud";
addTrustedUser "jac" "jac";

// define the methods where authentication must perform
restrict Account credit;
restrict Account withdraw;
```

Figure 4. The authentication.acc configuration file.

```
// define Account as a persistent root
configureClass Account root;

// account0 is a name of an object that should never
// change and that should be bound to the corresponding
// stored object
registerStatic account0;

// the storage configuration
configureStorage jac.aspects.persistence.PostgresStorage {
    // the database name, the user, and the password
    "account", "jac", ""
};
```

Figure 5. The persistence.acc configuration file.

root, i.e. all the referenced objects are also persistent. A static name is a name that never changes and that is stored within the persistent storage (here the *PostgresStorage*).

Once all the aspects have been applied and configured, the account instance is wrapped by a wrapping chain that extends its behavior. For instance, the authentication wrapper requires the username and the password before each call of the `credit` or `withdraw` methods. Moreover, the `Account` class is extended with meta-data as shown in figure 6. These extensions are made through the RTTI, as depicted in section 3.4.

Note that JAC also provides an IDE that facilitates the programmer's task by providing UML modeling facilities. The IDE is itself programmed in JAC and is a good sample of what

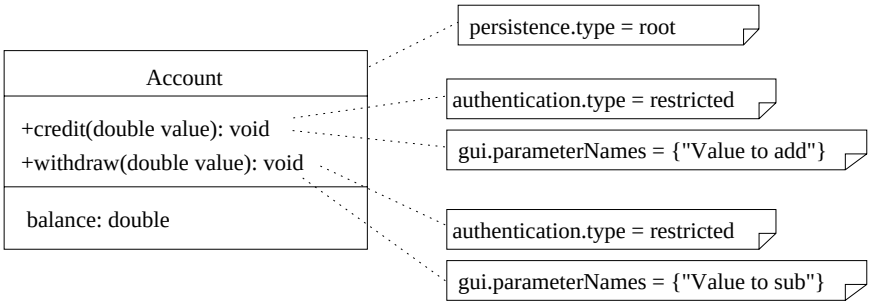

Figure 6. The *Account* class extended semantics with the three over-depicted aspects.

can be achieved in JAC. Figure 7 shows the programming of the Account application within the JAC's IDE.

4.3. Creating distributed applications and aspects

This section introduces distributed programming with JAC, first by showing how the existing deployment and consistency aspects can be woven into a given application, and next by programming an aspect component implementing a load-balancing concern. Further ahead in this article, section 5.3 explains how distributed aspects are implemented.

Deployment and consistency aspect components

Let us take a client/server application that consists of a server object (class `Document`, instance `document0`) that serves documents to some clients. The document server implements a method `int[] searchFor(String s)` that returns the indexes of the pages where one or several occurrences of the string *s* can be found. Client requests for the `searchFor` method can easily overload the server host when the number of pages is high. Hence, after describing how `document0` can be replicated on several hosts, we show how these client requests can be load-balanced.

If one considers the initial server located on host *s0* (left part of figure 8), then the idea is to replicate it on hosts *s1* and *s2*. JAC comes with a ready to use deployment AC. This AC provides two main methods: `deploy` and `replicate`. Both work with an existing instance, and either remotely deploy it on a given host, or remotely clone it on several hosts. The original instance is unchanged. The remote copies share the same local name as the original one (here `document0`). If this local name already exists on a host, an exception is thrown. By simply writing a configuration file for this AC, we can replicate `document0` on hosts *s1* and *s2*:

```
// deployment.acc
```

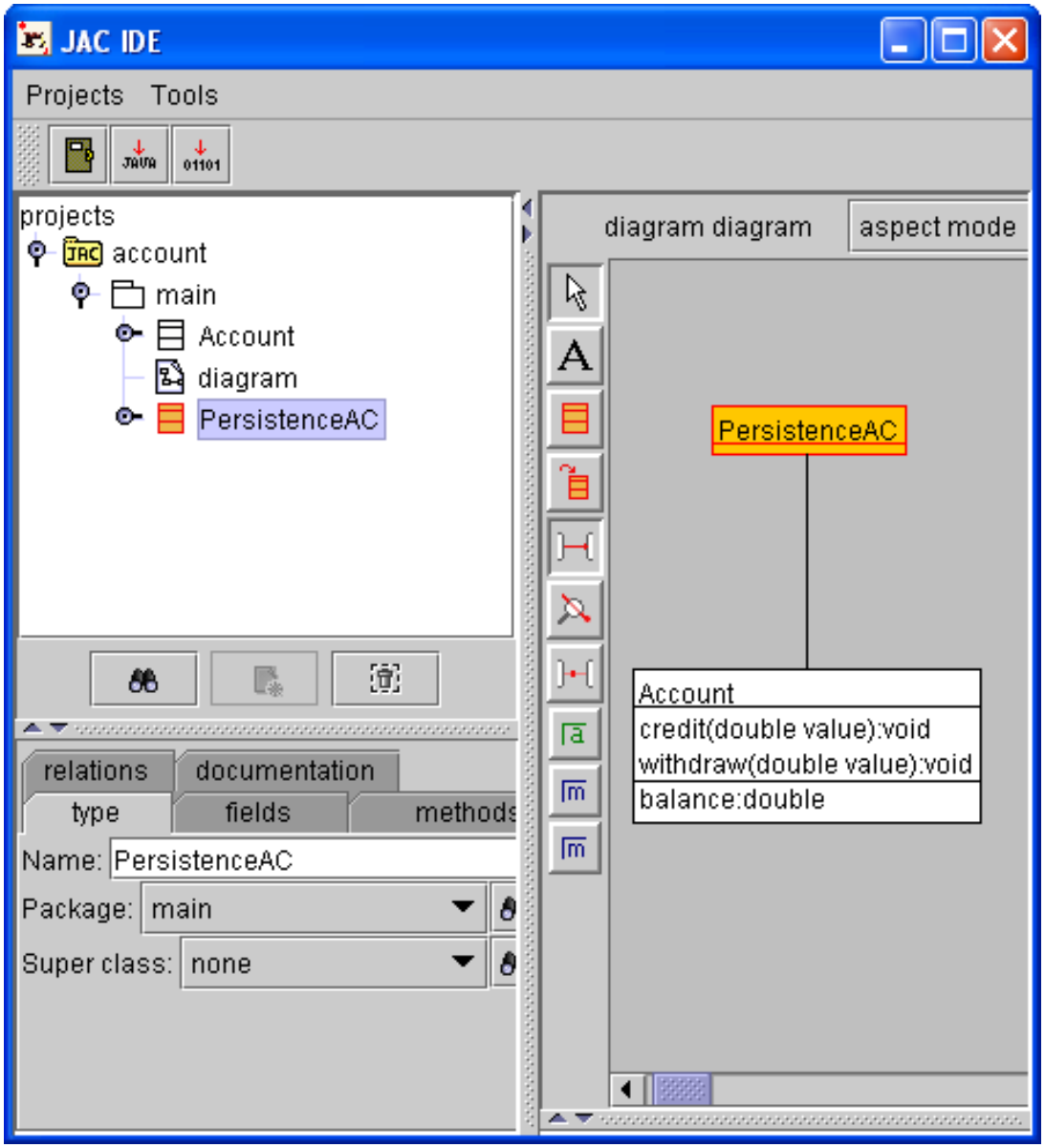

Figure 7. Programming an application with the JAC's IDE.

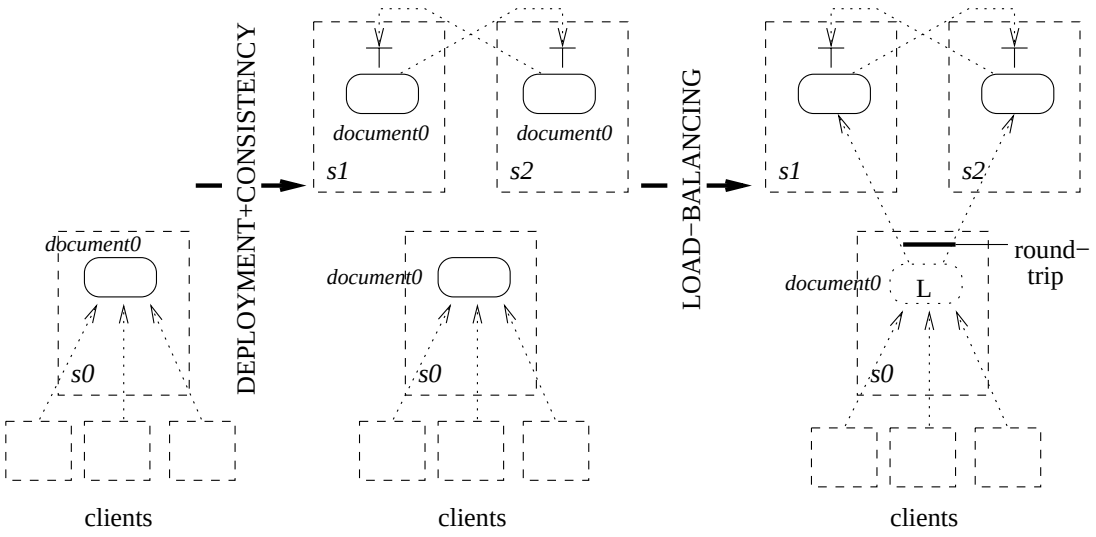

Figure 8. The document server application with a distribution aspect and a load-balancing aspect.

```
replicate "document0" "s1 || s2"
```

The second parameter of the `replicate` method is a host pointcut expression (see section 3.1.2). Here, we specify that both hosts *s1* and *s2* match the pointcut. A common concern is to keep these replicas in strong consistency so that updates (not shown here) do not lead to divergent replicas (note that the initial server is not part of the consistency). The `addStrongPushConsistency` method from the consistency aspect furnished by JAC allows to keep these replicas consistent. It takes two parameters: the instance, and a host pointcut expression designating the hosts where the replicas have to be kept consistent. The resulting distributed program is shown in the center part of figure 8.

```
// consistency.acc
addStrongPushConsistency "document0" "s1 || s2"
```

Load-balancing aspect components

Load-balancing the client requests can be achieved by wrapping the `searchFor` method of `document0` and by redirecting the calls to one of the two replicas. The corresponding AC is shown below. The pointcut specifies that method `searchFor` of instance `document0` (class `Document`) located on host `s0` is wrapped. The wrapper relies on two classes, not introduced before: `jac.core.dist.RemoteRef` and `jac.core.dist.Topology`. The former designates a

remote reference to a JAC object, whereas the latter represents the network topology on which the application is deployed (here *s1* and *s2*). The `invoke` method on the former class remotely invokes the referenced object. The latter class provides some utility methods such as `getReplicas` that returns the remote references of JAC objects that share the same local name (e.g. `document0`). The wrapping method `loadBalance` simply applies a round-robin algorithm.

```
import jac.core.Wrapper;
import jac.core.AspectComponent;
import jac.core.Interaction;
import jac.core.dist.RemoteRef;
import jac.core.dist.Topology;

public class LoadBalancingAC extends AspectComponent {

  public LoadBalancingAC() {
     pointcut(
        "document0", "Document", "searchFor", "s0",
        new LoadBalancingWrapper("document0"), "loadBalance"
     );
  }

  public class LoadBalancingWrapper extends Wrapper {
     int count = 0;
     RemoteRef[] replicaRefs;

     public LoadBalancingWrapper( String name ) {
        replicaRefs = Topology.get().getReplicas(name);
     }
     public Object loadBalance( Interaction i ) {
        if ( count >= replicaRefs.length )   count=0;
        return replicaRefs[count++].invoke(i.method,i.args);
     }
  }
}
```

Several other useful aspects can be programmed using the same technique such as fault-tolerance, caching (with a cache proxy), or broadcasting. Several sample implementations are provided in the `jac.aspects.distribution` package of JAC [28].

4.4. Implementing dynamic adaptability with aspects

In section 4.3, we have shown how to program a distributed AO application. In this section, we explain how dynamic adaptability can be performed. If we consider again the document server example, one can see that a strong consistency protocol has been chosen. However, if

consistency becomes less important, a weaker protocol can be used. For instance, instead of pushing all new versions of the data each time an update is performed, one may: (i) invalidate data on replicas if an update is performed, (ii) retrieve the new version of the data on the master replica (the one where the update has been performed) if a read operation occurs, (iii) work with this retrieved version until the next invalidation. Such a protocol saves unnecessary transmission of write operations. The `addWeakPullConsistency` method of the consistency AC implements it. The following file defines a configuration of this AC for `document0`.

```
// consistency-alt.acc
addWeakPullConsistency "document0" "s1 || s2"
```

By using the *consistency-alt* configuration, the network traffic will be considerably lowered when users change the document's data. However, when the available bandwidth is high, we prefer to use the strong consistency. Thus, a specific "agent" can be used to swap configurations depending on the available bandwidth. The following code uses the JAC API to implement such a behavior. Such a code assumes that there is no pending request while the policy is switched.

```
public class ConsistencyPolicySwapper {
  // .... constants definition, watching thread construction
  int policy = STRONG;
  public void watch() {
    if ( Network.availableBandwidth()>50 && policy==WEAK ) {
      setPolicy("consistency.acc");
    } else if ( Network.availableBandwidth()<=50 && policy==STRONG ) {
      setPolicy("consistency-alt.acc");
    }
  }
  void setPolicy(String accName) {
      Application app = ApplicationRepository.get().getApplication("document");
      // remove the consistency aspect for the document application
      // set the new configuration file
      // weave again the consistency
      ApplicationRepository.get().unextend("document","consistency");
      app.getAcConfiguration("consistency").setURL(new URL(accName));
      ApplicationRepository.get().extend("document","consistency");
}
```

5. The JAC architecture

In sections 3 and 4, we presented the programming model of JAC and introduced aspect components, pointcuts, and dynamic wrappers semantics. The goal of section 5 is to show how these different elements interact within the JAC architecture as well as to present some of the implementation issues.

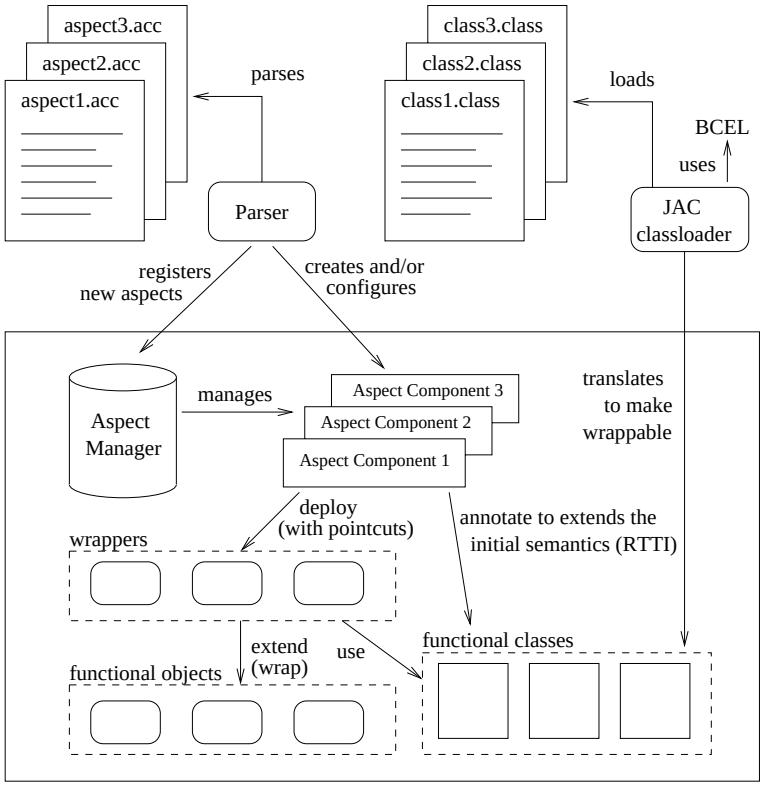

Figure 9. The JAC architecture: global flow of control of a JAC application.

5.1. Overview

Figure 9 shows how the different JAC system objects interact with the application objects to implement the aspect semantics depicted in section 3. On the right side of the figure, one can see that the functional classes are modified at the bytecode-level in order to make their instances wrappable. The bytecode translation mechanism is implemented with BCEL [29] and is further explained in section 5.2. Once the classes are translated, they are ready to be wrapped by the aspect components. When an aspect is woven into a given application, the JAC system first reads the available AC configuration (`*.acc` files on the upper left of the figure). The parser then invokes a set of configuration methods on the newly instantiated AC. These invocations trigger the creation of pointcuts and the tagging of the functional classes with some meta-data (through the RTTI).

When a new instance is created, the AC-Manager (aspect component manager) automatically notifies all the registered aspects so that the pointcuts wrap its methods according to the aspect configuration. The important point is that any AC can be woven or unwoven at any time. Moreover, its configuration file can be parsed again while the application is running. When an aspect is unwoven or when a new configuration is read, all the added meta-data and the pointcuts are destroyed by the system. This is possible due to the wrapping mechanism implementation for dynamic wrapping/unwrapping. The next section shows how the wrapping mechanism is implemented.

5.2. Dynamic wrapping implementation

JAC wrappers can be added or removed at runtime. The goal of this section is to explain and evaluate the underlying mechanisms that are used to implement this feature.

Contrary to regular wrappers that delegate to the wrappee and that implement the same interface as the wrappee [30, 31, 32], dynamic wrappers rely on a Meta-Object Protocol (MOP) [33, 34, 35, 36] that uses reflection for its implementation. The JAC MOP implementation uses a load-time transformation technique that inserts hooks into the wrappers. These hooks are reflective invocations that allow actual wrapping method to be resolved at runtime. Let us take a simple `C` class example.

```
public class C {
  public int m1(int i,String s) {
    // m1 body
  }
}
```

Our load-time transformation translates the class C into the following:

```
    import jac.core.Interaction;
    import jac.core.Wrappee;
    import jac.core.Wrapping;
    import jac.core.rtti.AbstractMethodItem;
    import jac.core.rtti.ClassRepository;

01> public class C implements Wrappee {
02>   static AbstractMethodItem _JAC_method_0;
03>   static {
04>     _JAC_method_0=ClassRepository.get().getClass("C")
05>       .getMethod("m1(int,String):int"); }
06>   List _JAC_wc_method_0;
07>   C() {
08>     _JAC_wc_method_0=Wrapping.getWrappingChain(this,_JAC_method_0);
09>   }
10>   // renamed method
11>   private int _org_m1(int i,String s) {
```

```
12>     // m1 body
13>   }
14>   // added stub method
15>   public int m1(int i,String s) {
16>     return ((Integer)Wrapping.nextWrapper(new Interaction(
17>       _JAC_wc_method_0,this,_JAC_method_0,
18>       new Object[]{new Integer(i),s},0)).intValue();
19>   }
20> }
```

The main translation consists of hiding the original method by renaming it: here `m1` is renamed into `_org_m1` (line 11). The original method is replaced by a stub method (same prototype, line 15) that wraps the original method by using `Wrapping.nextWrapper(Interaction)` (line 16). The fields `_JAC_method_0` (line 2) and `_JAC_wc_method_0` (line 6) are added for the purpose of optimization. `_JAC_method_0` caches the instance in the metamodel corresponding to the `m1` method, and `_JAC_wc_method_0` caches the wrapping chain (i.e. all the wrappers that currently wrap the wrappee). Pairs of `_JAC_method_`*i* and `_JAC_wc_method_`*i* fields are added for any other method defined in the original class. Wrapping or unwrapping a given object at runtime is accomplished by adding or removing an element in the method wrapping chain (i.e. field `_JAC_wc_method_0`) that can be retrieved with `Wrapping.getWrappingChain(wrappee,method)`.

The initial interaction is created in the stub method (lines 16-18). It takes the wrapping chain that is internally used, the wrappee's reference, the current method, the parameters (translated into an array of objects), and an initial rank in the wrapping chain (0). The following code shows the implementation of the `Wrapping.nextWrapper` method (the implementation also handles statics and constructors, following the same principles).

```
public static Object nextWrapper(Interaction interaction) {
  try {
    // if the current rank is smaller than the wrapping chain's size
    // some wrappers still remain to call
    if (interaction.wrappingChain.size>interaction.rank) {
      // invoke the wrapper of current rank ([0]->wrapper,
      // [1]->wrapping method)
      return ((Method)interaction.wrappingChain.get(interaction.rank)[1])
        .invoke(interaction.wrappingChain.get(interaction.rank)[0],
                new Object[]{interaction});
    } else {
      // invoke the original method
      return ((MethodItem)interaction.method).getOrgMethod()
        .invoke(interaction.wrappee,interaction.args);
    }
  } catch (InvocationTargetException e) {
    [...] // handle exception with exception handlers
  } }
```

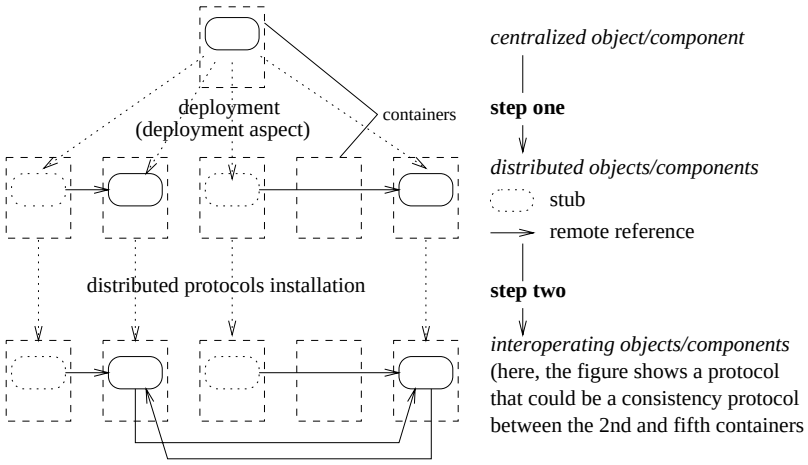

Figure 10. The distribution mechanism in JAC: focus on a single object.

Finally, the `proceed(interaction)` method defined in section is simply equivalent to `Wrapping.nextWrapper(interaction.rank+=1)`.

5.3. Distribution

As mentioned in section 4.3, JAC handles the distribution of aspects. Our motivation for this feature is that, when programming distributed applications, some aspect properties may be required on several containers where the distributed application is running so that the aspect modifications crosscut objects that are not necessarily located in a single container. As simple examples, when applying a tracing aspect to a distributed application, most of the programmers would expect all the calls on all the objects to be traced, whether they are local or remote. In client/server schemes, an authentication aspect should be able to add the authentication concern, on all the clients and all the server containers. Finally, several aspects such as data consistency or load-balancing are inherently distributed and need to be aware of distribution information.

As shown in figure 10, the core distribution mechanism in JAC is based on two kinds of ACs.

- A deployment AC is used to distribute objects (step one). This AC defines a pointcut that wraps the constructors to perform the deployment.
- A set of aspects that implement distributed protocols (step two) which allow the different objects of the application to collaborate between the different containers where the application is deployed.

The deployment AC relies on a daemon program that runs on all the hosts where the distributed application is to be run. This daemon provides a remotely accessible interface that comes with two personalities: Java/RMI and CORBA. Hence, daemons communicate and are accessible with either of these two middlewares. The remote interface of the daemon performs three basic tasks: remote uploading of class and AC bytecode, remote instantiations of classes and ACs, and remote copy of objects and AC instances. As illustrated in the first step of figure 10, an instance located on a host can be remotely instanciated, and stubs to access these copies can be installed on other hosts. Notice that this technique does not intend to provide some form of migration support for already executing objects and aspects. No mechanism is provided to stop a running thread and to restart it on another host. This is primarily intended to be a deployment technique to install a distributed application. If developers use it to migrate a running application, they have to take care of managing threads by themselves.

Besides this deployment aspect, we provide a set of distributed protocols implemented as ACs. They implement various schemes for data consistency, multicasting, load-balancing with a round-robin algorithm, and fault tolerance with active replication.

As explained in section 5.1, the AC-manager handles ACs woven onto an application. ACs are registered in the AC-manager when they are woven, and unregistered when they are unwoven. ACs can then be seen as data handled by the AC-manager. As any other object, the AC-manager can be remotely deployed, replicated, and associated with a consistency protocol (top part of figure 11). When an aspect is woven or unwoven on a site, the state of the AC-manager is modified, and the consistency protocol propagates this modification on the other sites (middle part of figure 11). Hence, AC instances are replicated on all the hosts of the environment, and pointcut semantics is made distributed (bottom part of figure 11).

5.4. Performance measurements

The critical point of the JAC framework in terms of performance is the dynamic wrapper invocation mechanism. Since this invocation relies on reflection to achieve dynamic adding or removing of aspects, the performance overhead of JAC arises primarily from the reflective call's overhead (itself mainly arising from the array of objects construction passed as argument)[‡].

Table IV shows the performance of empty method calls on regular objects and on JAC wrappable objects. These tests are performed with a bench program that calls several methods with different prototypes and is available in the JAC distribution [37]. The bench program was run under Linux with a Pentium III 600 MHz with 256KB of cache and with the SUN's Java HotSpot Client VM version 1.4.

One can see that a call on a JAC wrappable object is comparable to a reflective call on a regular Java object (with an overhead of 29%). Each time a wrapper is added, an overhead of about 40% of the initial time is added (note that the bench adds empty wrappers that just call `proceed` in their implementations).

[‡]Note that the reflective invocation is not yet fully optimized in Java. We assume that it will be greatly enhanced in future versions since it exists techniques to do so.

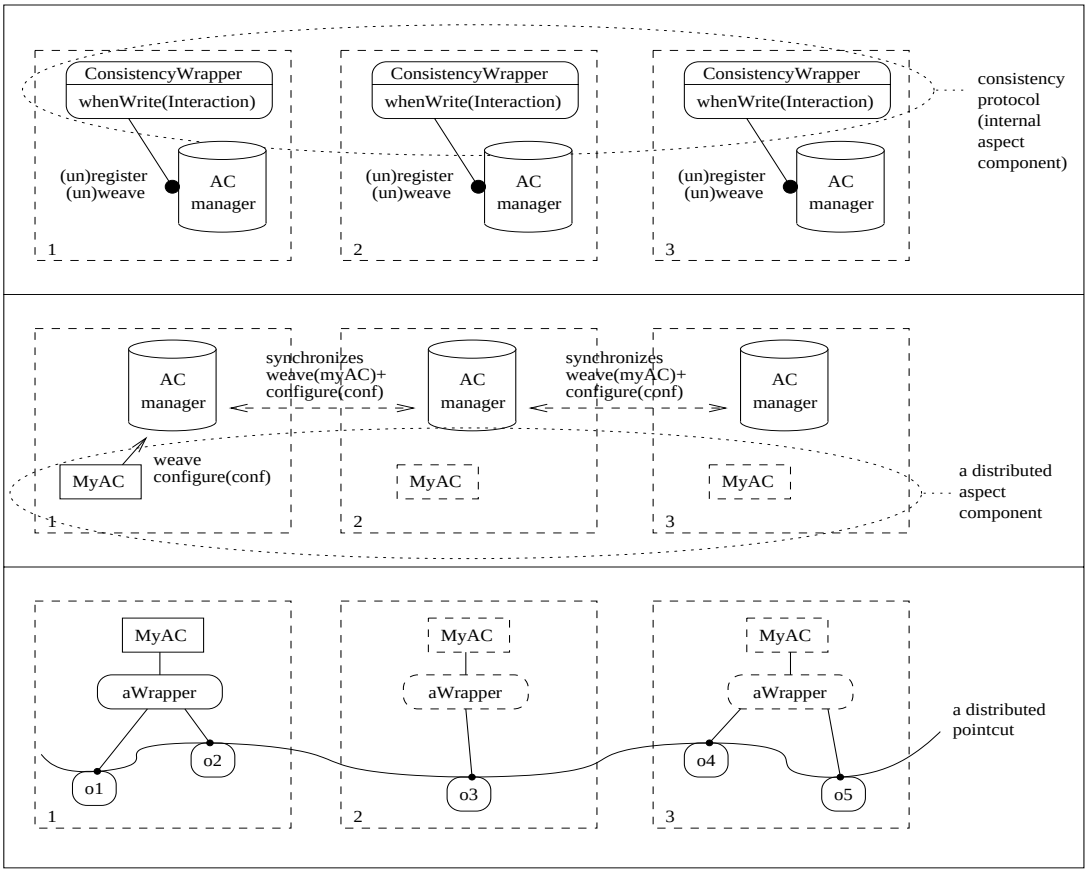

Figure 11. The distribution of aspects mechanism.

Finally, the cost of adaptability is quite high (as for reflection) compared to compiled approaches such as AspectJ. However, with real-word aspects and especially when the application is distributed, this cost becomes negligible. For instance, a servlet invocation costs around 1 ms in the same environment. Moreover a transactional distributed call to an EJB is approximatively 30 ms, which is much slower than a JAC stand-alone object.

For the moment, the JAC approach is well-suited for middle-grained wrappable objects (only business objects are made wrappable in real-word applications, technical components that require better performance are not aspectized) and for distributed and adaptable programming.

Table IV. Comparative performance measurements for Java and JAC.

Type of calls	Number of calls	Total time	Time per call	Overhead
(A) regular object calls	6,000,000	55 ms	9.16 ns	-
(B) reflective calls	60,000	47 ms	0.78 μs	(A)x 85
(C) JAC objects calls (0 wrapper)	60,000	61 ms	1 μs	(B)x 1.29
JAC (1 wrapper)	60,000	85 ms	1.41 μs	(C)+41%
JAC (2 wrappers)	60,000	110 ms	1.83 μs	(C)+83%
JAC (3 wrappers)	60,000	130 ms	2.16 μs	(C)+116%

6. Related works

This section compares JAC with existing approaches. The research community is actively working on AOP, and many works may deserve being presented. The `aosd.net` web site, which is a major source of information for AOP, mention more than 30 AOP-related projects. We choose to focus on the ones that provide languages, or tools for AOP. Hence, even if they are of great interest, we leave apart works on AOP in the design and analysis stages [14][15], UML-based design notations for aspects [16][17][18], AOP for databases [38], AOP for operating systems [39], AOP for component based systems [40][41], formalization of AOP systems [42], OO frameworks to support crosscutting concerns [43], or AOP for other languages than Java.

The projects we review are classified according to the way aspects are introduced: either at compile time (AspectJ, CFOM, Aspectual components), or at run time (PROSE, Lasagne) with some support for dynamic AOP.

Compile time AOP

AspectJ [12] is an extension of Java that provides support for the implementation of crosscutting concerns through pointcuts (collections of principle points in the execution of a program), and advices (method-like structures attached to pointcuts). Precedence rules are defined when more than one advices applies at a join point. In many features (e.g. pointcut definitions) AspectJ has a rich and vast semantics that allows the programmer to implement aspects similarly to JAC but at compile time (with better type checking and development tool support).

The composition filter object model (CFOM) [5] is an extension to the conventional object model where input and output filters can be defined to handle sending and receiving of messages. This model is implemented for several languages, including Smalltalk, C++ and Java. The latter implementation is an extension to the regular Java syntax where keywords are added to declare, for instance, filters attached to classes. The goals of this model and ours

are rather similar: to handle separation of concerns at a meta level. Nevertheless, JAC does not require any language extension.

Aspectual components [6] and their direct predecessors adaptative plug and play components [44, 45] define patterns of interaction, called participant graphs (PG), that implement aspects for applications. PGs contain participants roles (e.g. publishers and subscribers in a publish/subscribe interaction model) that, (1) expect features about the classes upon which they will be mapped, (2) may re-implement features, and (3) provide some local features. PGs are then mapped onto class graphs with entities called connectors, that define the way aspects and classes are composed. Aspectual components can be composed by connecting part of the expected interface of one component to part of the provided interface of another. Nevertheless, it seems that by doing so, the definition of the composition crosscuts the definition of the aspects, loosing by this way the expected benefits of AOP. Caesar [46] is the direct successor of these two projects and introduces the notion of collaboration interfaces (CI). A CI decouples the aspect implementation from the aspect binding. These two elements are them related by a specification of a communication protocol, a so-called CI. Caesar introduces some syntactic extensions to support these features, and a compiler is under development.

Dynamic AOP

The projects reviewed in this section are the ones closest to JAC. They implement some mechanisms for weaving and unweaving aspects at run time.

An interesting approach for dynamic AOP is the PROSE project [47] that implements the dynamic weaving mechanism by using the Java Just-In-Time compiler. PROSE implements some specific optimizations for efficient dynamic weaving that are not supported by JAC (e.g. type-based optimizations). In the future, we hope to be able to standardize the weaver API in order to be able to seamlessly reuse efficient weaving engines (or even compile-time weaving engines) when needed.

Another related approach is the Lasagne project [48] that provides dynamic weaving/unweaving of aspects that is similar to the JAC implementation. The dynamicity is achieved by policy selection on the client. The distribution mechanism is provided by a regular ORB which makes the actual distribution of the aspects difficult. In fact, the main difference between JAC and Lasagne arises from the distributed pointcut notion which is, as far as we know, original to the JAC framework.

The JAC approach for dynamic AOP is based on load-time class modification. Hence, a customized class loader is provided to perform these modifications. Another technique for dynamic AOP relies on the Java Platform Debugger Architecture (JPDA) [49]. The RtJAC [50] project uses this architecture to perform dynamic transformation of classes. The Wool [51] project also uses JPDA. Wool uses breakpoints to introduce hooks in a program and the ability provided by the HotSwap mechanism of JPDA, to reload classes.

Finally, several projects (JMangler, Javassist, BCEL, ASM) share many implementation issues with AOP and are in many cases used as building blocks for these systems. Frameworks such as JMangler [52], Javassist [53], BCEL [29] or ASM [54] provides APIs for bytecode manipulation. A base program can be modified to introduce new concerns. These tools provide the basis on which AOP frameworks such as JAC are built. Earlier versions of JAC used

Javassist, we now use BCEL. Bytecode manipulation and instrumentation is a very general technique that is needed in many application domains. The upcoming release of J2SE 1.5, code named Tigger, will start to address this problem with a new `java.instrument` API.

7. Conclusion

This paper introduces JAC, a framework for AOP in Java. We present its key features: dynamic weaving, distribution of aspects, and aspect re-use through configuration. We also give several indications on how to use the framework and focused on some implementation points.

The key concept introduced by JAC is the notion of Aspect Component (AC). An AC is the software entity that captures a crosscutting concern. The pointcut mechanism associated with an AC allows, like in other approaches such as AspectJ, expresses the elements of the base objects where the aspect is to be woven. The originality of our approach is that this mechanism can be applied to distributed objects: a crosscut can modify the semantics of objects physically located on distributed hosts. To achieve this, JAC comes with a container mechanism. Our containers host both business objects and ACs' instances. They are remotely accessible using either CORBA or RMI software buses.

Moreover, ACs can be dynamically (un)woven to the application. This dynamicity enables application adaptability which can be extremely useful within distributed and changing environments.

Besides ACs, JAC also provides a powerful mechanism for configuring existing ones. Based on methods defined in ACs, customized configuration files can be provided for each aspect involved in a given application. The idea is to let the person in charge of software integration express in a declarative way the steps required to configure an application. This mechanism can be related to a Domain Specific Language (DSL) for aspect configuration.

Currently, we are investigating the possibility of standardizing the weaver API in order to re-use other weaving engines that can be more suitable for specific requirements (e.g. the JIT-based Prose engine [47]). The first versions of this API is available on the AOP Alliance project web site [55].

In the near future, we also intend to provide better support for aspect configuration which is, to us, one of the central points of AOP. The challenge is to be able to re-use existing aspects. The existing aspect configuration language syntax could be upgraded. Support for other forms of configuration (e.g. with XML files) could be provided. Moreover, the pointcut grammar should also be more rigorously defined in order to support compilation and validation checks.

REFERENCES

1. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
2. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
3. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

4. L. Bergmans M. Aksit and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of ECOOP'92*, volume 615 of *Lecture Notes in Computer Science*, pages 372–395. Springer-Verlag, 1990.
5. L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *CACM*, 44(10):51–57, 2001.
6. K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report Technical Report NU-CCS-99-01, Northeastern University's College of Computer Science, avril 1999.
7. W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA'93, volume 28 of SIGPLAN Notices*, pages 411–428, octobre 1993.
8. H. Ossher, K. Kaplan, W. Harrison, A. Matz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA'95*, volume 30 of *Sigplan Notices*, pages 235–250. ACM Press, 1995.
9. H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3), 1996.
10. H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.
11. H. Ossher and P. Tarr. *Multi-dimensional separation of concerns and the hyperspace approach*, chapter Software Architectures and Component Technology: The State of the Art in Research and Practice. In L. Bergmans and M. Aksit, kluwer academic publishers edition, 2001.
12. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
13. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, June 2001.
14. M. Katara and S. Katz. Architectural views of aspects. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 1–10. ACM Press, 2003.
15. Awais Rashid, Ana Moreira, and Joao Arajo. Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 11–20. ACM Press, 2003.
16. J. Suzuki and Y. Yamamoto. References extending UML with aspects: Aspect support in the design phase. In *3rd AOP Workshop at ECOOP*, 1999.
17. S. Clarke and R. J. Walker. Towards a standard design language for AOSD. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 113–119. ACM Press, 2002.
18. D. Stein, S. Hanenberg, and R. Unland. A UML-based aspect-oriented design notation for AspectJ. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 106–112. ACM Press, 2002.
19. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Dynamic wrappers: handling the composition issue with jac. In *Proceedings of TOOLS USA 2001*, 2001.
20. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In *Proceedings of Reflection 2001*, LNCS 2192, pages 1–21, May 2001.
21. K. Czarnecki and U. Eisenecker. *Generative Programming*, chapter Aspect-Oriented Programming. Addison-Wesley, 2000.
22. T. Elrad, R. Filman, and A. Batef. Aspect-oriented programming. *Communications of the ACM*, 44(10):51–57, 2001.
23. Special issue on Aspect-Oriented Programming. Communications of the ACM, 44(10), 2001.
24. R. Pawlak, L. Duchien, G. Florin, L. Martelli, and L. Seinturier. Distributed separation with aspect components. In *Proceedings of TOOLS Europe 2000*, June 2000.
25. Sun Microsystems. *Enterprise Java Beans White Paper*, December 1998. `http://www.javasoft.com/products/ejb/`.
26. R. Pawlak, L. Duchien, and G. Florin. An automatic aspect weaver with a reflective programming language. In *Proceedings of Reflection'99*, July 1999.
27. S. Chiba. Open C++ release 1.2 programmer's guide. Technical Report 9303, Department of Information Science, University of Tokyo, 1993. `ftp://ftp.is.s.u-tokyo.ac.jp/pub/techreports/TR93-03-letter.ps.Z`.
28. R. Pawlak, L. Martelli, and L. Seinturier. The JAC API. `http://jac.objectweb.org/docs/javadoc/`.
29. The Jakarta Project. Bcel. `http://jakarta.apache.org/bcel/`.
30. J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In *Prodeedings of ECOOP'98*, 1998.

31. G. Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of the ECOOP'99*, volume 1628 of *Lecture Notes in Computer Science*, 1999.
32. M. Buchi and W. Weck. Generic wrappers. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 201–225. Springer, June 2000.
33. P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the 2nd Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*, volume 22 of *SIGPLAN Notices*, pages 147–155. ACM Press, December 1987.
34. G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
35. C. Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.
36. G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.
37. R. Pawlak, L. Martelli, and L. Seinturier. The JAC project home page. `http://jac.objectweb.org`.
38. A. Rashid. A hybrid approach to separation of concerns: The story of sades. In *Proceedings of Reflection 2001*, LNCS 2192, pages 231–249, May 2001.
39. Y. Coady and G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59. ACM Press, 2003.
40. F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 65–75. ACM Press, 2002.
41. D. Suve, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM Press, 2003.
42. R. Douence, O. Motelet, and M. Sudholt. A formal definition of crosscut. In *Procedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186. Springer, September 2001.
43. C. Constantinides, T. Elrad, and M. Fayad. Extending the object model to provide explicit support for crosscutting concerns. *Software Practice and Experiences*, 32(7):703–734, June 2002.
44. M. Mezini and K. Lieberherr. Adaptative plug-and-play components for evolutionary software development. In *Proceedings of OOPSLA'98*, volume 33 of *SIGPLAN Notices*, pages 96–116. ACM Press, 1998.
45. M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In L. Bergmans and M. Aksit, editors, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2000.
46. M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, 2003.
47. A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: Efficient dynamic weaving for java. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 100–109. ACM Press, 2003.
48. E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, Joergensen, and N. Bo. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of ICSE'01*, 2001.
49. Sun Microsystems. *Java Platform Debugger Architecture*, 2002. `http://java.sun.com/j2se/1.4/docs/guide/jpda/index.html`.
50. M. Espak. Improving efficiency by weaving at run-time. In *Proceedings of the Young Researchers Workshop at the 2nd International Conference on Generative Programming and Component Engineering (YRW/GPCE'03)*. `http://se.inf.ethz.ch/events/gpce_yrw03/`.
51. Y. Sato, S. Chiba, and M. Tatsubori. A selective just-in-time aspect weaver. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, volume 2830 of *Lecture Notes in Computer Science*, pages 189–208. Springer, 2003.
52. G. Kniesel, P. Costanza, and M. Austermann. JMangler - A Framework for Load-Time transformation of Java class files. In *IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, 2001.
53. S. Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, June 2000.
54. E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *Journes Composants 2002 (JC'02)*, November 2002. `http://asm.objectweb.org/current/asm-eng.pdf`.
55. The AOP Alliance web site. `http://sourceforge.net/projects/aopalliance`.

8. Appendix

Allowed keywords or expressions within the pointcut expressions.

Pointcut Type	Keyword / expression	Semantics
class	ALL	all the classes
	¡name¿+	all the child classes of *name*
	¡name¿-	all the parent classes of *name*
object	ALL	all the objects (instances of the class expr)
	OPath expr	any object in relation with the root object
method	ALL	all the methods
	STATICS	all the static methods
	CONSTRUCTORS	all the constructors
	MODIFIERS	all the methods that modify the object's state
	ACCESSORS	all the methods that read the object's state
	GETTERS[(...)]	the getters
	SETTERS[(...)]	the setters
	ADDERS[(...)]	the methods that add an object to a collection
	REMOVERS[(...)]	the methods that remove an object to a collection
	FIELDGETTERS	all the getters for primitive fields
	FIELDSETTERS	all the setters for primitive fields
	REFGETTERS	all the reference getters
	REFSETTERS	all the reference setters
	COLGETTERS	all the collection getters
	COLSETTERS	all the collection setters